浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：范源颢

学　　院：计算机科学与技术学院

　　系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3180103574

指导教师：陈文智

2020 年　12 月 21　日

浙江大学实验报告

课程名称：计算机体系结构 实验类型：综合

实验项目名称：带控制冒险解决策略的 CPU 流水线

学生姓名：范源颢 专业：计算机科学技术 学号：3180103574

同组学生姓名：周寒靖 指导老师：陈文智

实验地点：曹光彪西楼-301 实验日期：2020 年 12 月 21 日

一、 实验目的和要求

目的：

1. 了解控制冒险出现的原因；
2. 掌握解决控制冒险问题的若干方法：
   a) 冻结或冲刷
   b) 预测未选中
   c) 预测选中
   d) 延迟分支
3. 掌握停顿一周期的预测未选中和预测选中分支方法。
4. 掌握流水线旁路无效而必须使用流水线停顿的情况
5. 掌握利用程序验证“带控制冒险处理的流水线 CPU”的方法

要求：

1. 设计 CPU 控制器和数据通路，组合这些基本单元形成一个单周期 CPU。
2. 用程序验证单周期 CPU，观察程序的执行情况。

二、 实验内容和原理

2.1 我们在本次试验中依旧实现 16 指令的 MIPS 汇编语言。

各条语句的语法和含义如下：

16 MIPS Instructions

Bit #	31..26	25..21	20..16	15..11	10..6	5..0	Operations	
R-type	op	rs	rt	rd	sa	func		
add		rs	rt	rd	00000	100000	rd = rs + rt; with overflow	PC+=4
sub		rs	rt	rd	00000	100010	rd = rs - rt; with overflow	PC+=4
and		rs	rt	rd	00000	100100	rd = rs & rt;	PC+=4
or	000000	rs	rt	rd	00000	100101	rd = rs | rt;	PC+=4
sll		00000	rt	rd	sa	000000	rd = rt << sa;	PC+=4
srl		00000	rt	rd	sa	000010	rd = rt >> sa (logical);	PC+=4
slt		rs	rt	rd	00000	101010	if(rs < rt)rd = 1; else rd = 0; <(signed)	PC+=4
jr		rs	00000	00000	00000	001000		PC=rs
I-type	op	rs	rt		immediate			
addi	001000	rs	rt		imm		rt = rs + (sign_extend)imm; with overflow	PC+=4
andi	001100	rs	rt		imm		rt = rs & (zero_extend)imm;	PC+=4
ori	001101	rs	rt		imm		rt = rs | (zero_extend)imm;	PC+=4
lw	100011	rs	rt		imm		rt = memory[rs + (sign_extend)imm];	PC+=4
sw	101011	rs	rt		imm		memory[rs + (sign_extend)imm] <-- rt;	PC+=4
beq	000100	rs	rt		imm		if (rs == rt) PC+=4 + (sign_extend)imm <<2;	PC+=4
J-type	op			address				
j	000010			address			PC = (PC+4)[31..28],address<<2	
jal	000011			address			PC = (PC+4)[31..28],address<<2 ; $31 = PC+4	

2.2 在本次实验中，我们只需要改进我们之前的 5 阶段流水线 CPU 设计，实现一个“1 周期停顿的预测选中”策略。为此，我们需要 1.将计算分支条件和分支跳转地址的部件提前若干周期完成；2.将旁路单元的功能提前若干周期完成。我们在具体实现时，会使用这种方法，但是对于控制冒险的理论学习中，我们也会涉及到其他方法。

2.3 控制冒险的定义

控制冒险来源于流水线工作中的分支操作和其他改变 PC 寄存器（Program Counter，程序计数器或者程序指针寄存器）值的操作。因为存在分支操作，所以我们下一条指令具体是哪一条是不确定的，因此，我们可能需要停顿等待。

然而，这对于流水线整体的效率提升非常不利，通常情况下，控制冒险造成的性能损失会比上节中的数据冒险更大（既是因为分支冒险难以处理，也是因为分支指令出现的情况比数据冒险出现的情况更多）。

为此，我们需要减少流水线分支的代价。我们有如上文 1.2.a)-d)的四种方法减少代价，我们将在下文介绍。但是注意，我们这些处理机制是静态的，也就是说，对于程序中执行的每条分支指令，我们都采用不变的方法处理这些指令避免流水线出现分支冒险，这样的策略比较简单，我们也可以使用更加复杂的软硬件技术，使得我们对于不同的分支指令有不同额的处理方法，这种策略就是动态处理方法。

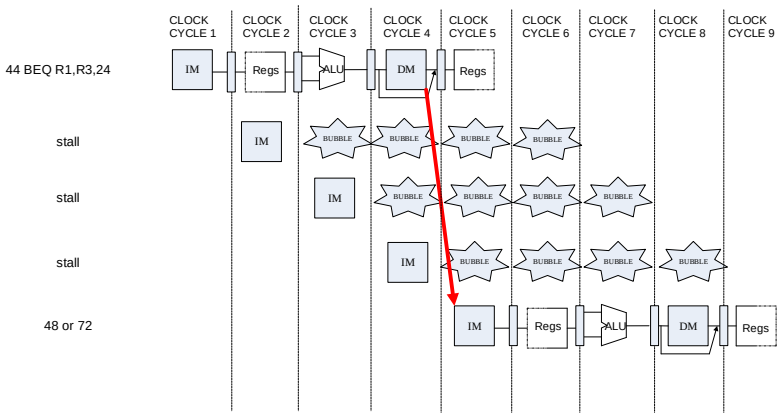
我们使用如下所示的例程来分析问题：

	Address	Instruction
	36	Nop
	40	Add R30,R30,R30
Branch to 72→	44	Beq R1,R3,24
if R1!=R3, it executes in sequence	48	And R12,R2,R5
	52	Or R13,R6,R2
	56	And R14,R2,R2
	60	
	64	
	68	
If R1=R3, it executes these instruction	72	Lw R14,50(R7)
	76	

在第 44 条指令有一条跳转指令，假如我们的指令选择分支，那么下一条指令将跳转到 72 号，如果条件不满足，不选择分支，那么我们的程序将按照顺序执行 44，48，……。

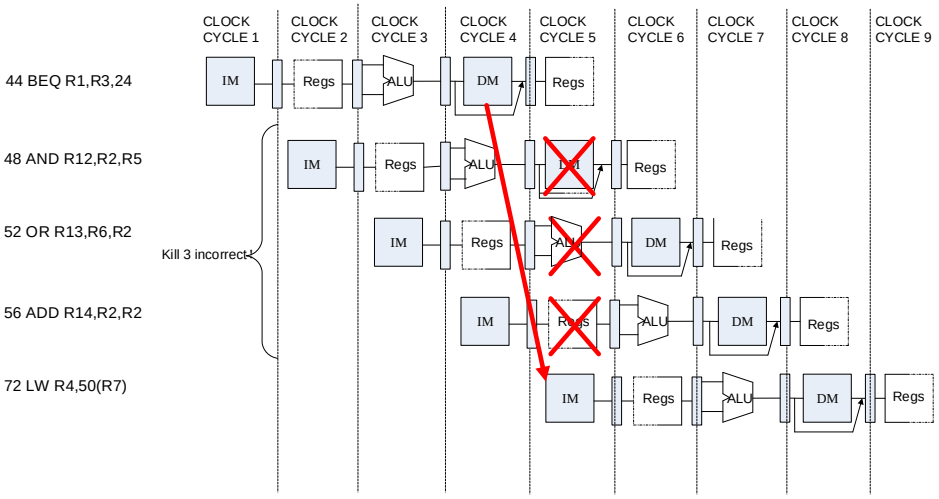
2.4 冻结和冲刷：

这是处理分支冒险的最简单的方法，我们遇到分支指令的时候，需要冻结流水线，然后等待到分支运算的结果在 MEM 阶段存入寄存器之后，再执行后续指令，这样我们可以保证程序的正常运行，这是一种处理分支冒险的策略，他是静态的，而且总可以保证指令的正确的执行，而且也不一定是最劣的，因为其他策略可能会涉及到指令的重做和撤回，而这种策略可以保证每一次的停顿可控。即便如此，他依旧是一种初等的策略，因为停顿的占比显然被控制的很多，这样浪费了很多 CPU 时间。（如下所示，这里引入了三个流水线停顿）



2.5 预测未选中策略

2.4 的效率是一个比较大的短板，一种性能更高但略微复杂的机制是将每个分支都看作一种未选中分支的情况，允许硬件按照流水线执行下一条指令，而对于预测未选中策略来说，我们是知道 CPU 应该选取的下一条指令是什么的，就是单纯的 PC+4，按照 add, sub 等一般指令的情况，取出下一条指令来进行 IF 处理就可以了，不过我们需要警惕，此时的分支结果尚且没有运算出来，因此有可能出现预测失败的状态，那么我们就必须撤回因为预测失败而从流水线中预作的几条指令，如果我们单纯在 MEM 阶段计算分支地址，那么我们就必须撤回至少三条指令，这样一来，在最坏的情况下，我们的策略甚至不如 2.4，因为我们还需要支付额外的撤销指令的时间。



2.6 预测选中策略

统计表明，大概有 60%的分支指令其分支会被选择，也就是会有一个地址跳转的操作，因此，我们可能进行预测指令会引发跳转更加节约时间。

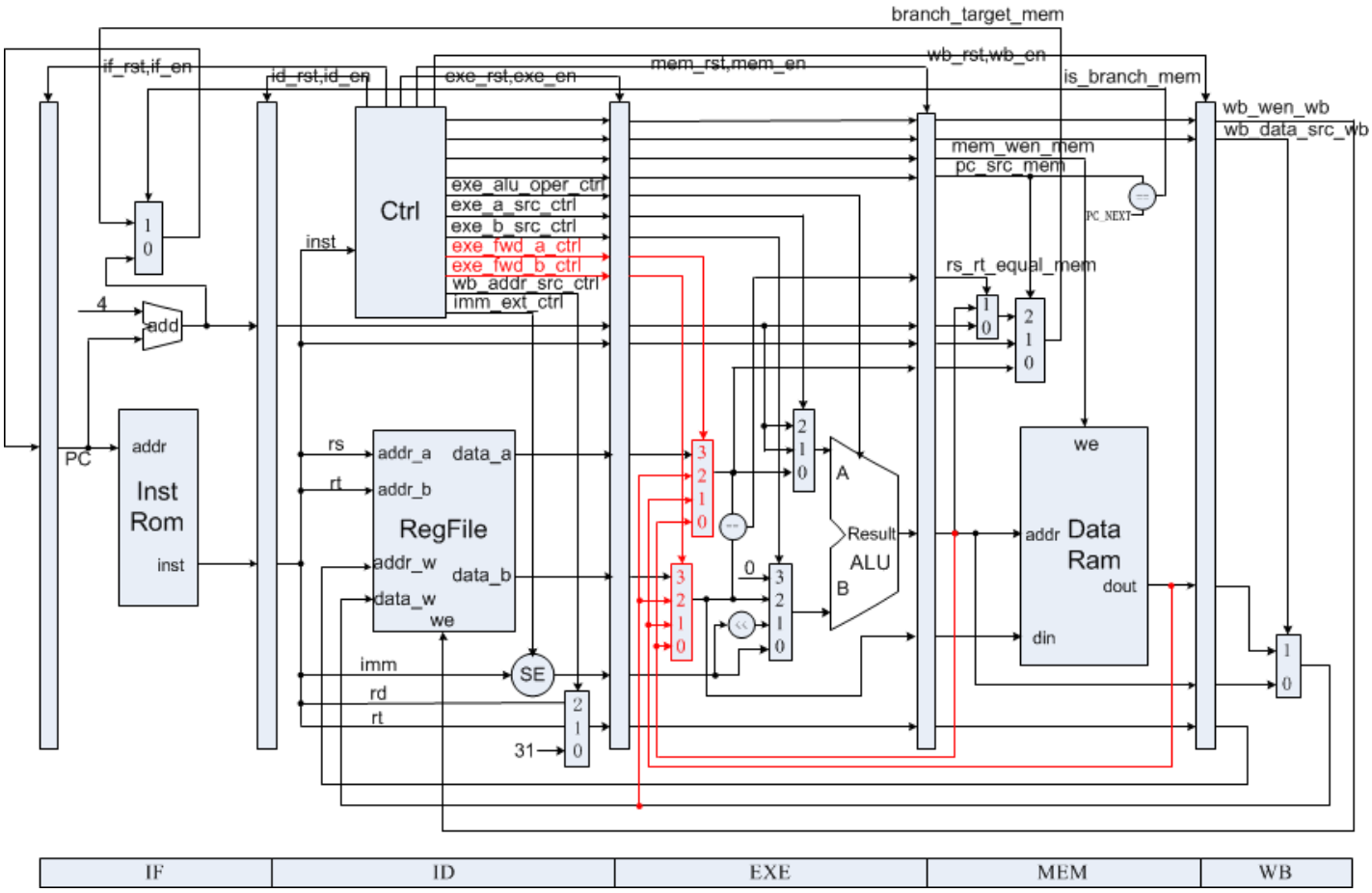
问题是，我们不知道我们跳转的地址，我们只有等待到分支地址被计算出来之后才可以执行这条指令的 IF，然后得到结果之后立即从指令寄存器中选取这条指令。初看起来，这种方法似乎和 2.4 没有区别

我们发现，其实我们可以把地址的计算进行前移，因为地址运算将是一个单纯的加法运算，他不涉及运算符的分析，于是我们只需要增加一个简单的加法器作为新增硬件，然后将分支地址的计算从 MEM 阶段迁移到 EXE 阶段，那么我们会发现，我们停顿的周期从三个减少到了两个。

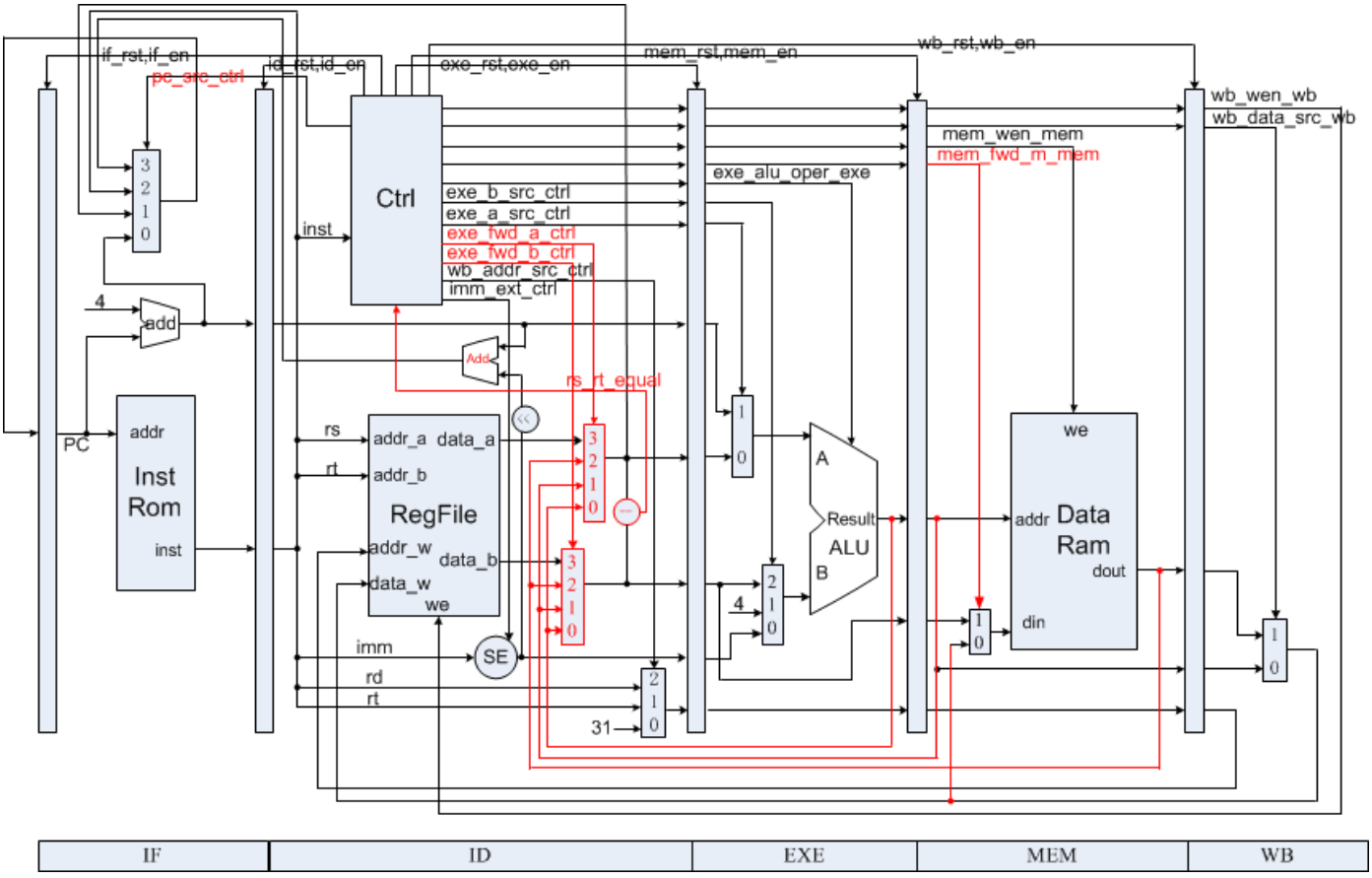
事实上，我们可以更进一步，将地址运算的过程直接从 EXE 阶段提前到 ID 阶段，因为 ID 阶段已经获得了运算跳转地址所需要的全部的立即数和寄存器的值，于是我们在这里增加一个加法器，直接运算出结果，同时，我们也可以在此时判断寄存器中的两个值是否相等，这样我们停顿的周期从两个又减小到了一个。

我们不能再将运算的过程提前了，因为 ID 之前的 IF 阶段时从指令寄存器中读取指令的过程，这一过程在结束之前我们都无法判定我们读取到的指令是否包含分支操作。

我们在前面说明的“提前”操作，是涉及到硬件上的变化的，具体而言，我们的 datapath 将从下图中的 origin datapath 变化为 current datapath，这样才能保证我们在 ID 阶段即可执行地址的运算。



origin datapath ↑



current datapath ↑

注意我们如果要前移比较寄存器值的阶段，那么我们必须也要将旁路的单元前移，因为硬件是复用的，在执行其他指令，解决数据冒险的时候，我们需要旁路的机制解决问题，于是旁路单元也便需要跟着前移，这样一来，旁路单元的输入也要跟着前移，所以我们看到的原来 MEM、WB 前移到了 EXE、MEM 阶段。

2.7 分支延迟

我们发现，我们无论如何都无法取消实验中一开始出现的那个停顿，我们成这样的停顿为分支延迟时隙，我们在时隙中不应该选择什么都不做，我们可以尝试做一些不影响程序正确性的操作，这种操作可以是循环体中的累加操作，也可以是使用的寄存器不再之后的程序引用的操作，在实际情况中，编译器在生成汇编指令的时候会充分考虑这种情况，将分支延迟时隙尽可能填满，这样我们程序总体的运行效率会更高，在本实验中，我们其实可以使用自己调整汇编代码的语句顺序，使得分支时隙得以被填满。

下图展示了我们新的汇编指令：

```
00000824 main:    and $1, $0, $0          # address of data[0]
34240050          ori $4, $1, 80          # address of data[0]
0c00000a call:    jal sum                 # call function
20050004          addi $5, $0, 4          # counter
ac820000 return:  sw $2, 0($4)            # store result
8c890000          lw $9, 0($4)            # check sw
ac890004          sw $9, 4($4)            # store result again
01244022          sub $8, $9, $4          # sub: $8 <- $9 - $4
08000008 finish:  j finish                # dead loop
00000000          nop                     # done
```

前半段，红色表示我们将 jal 指令前移了，这样 addi 表面上在 jal 之后，但是根据流水线原理，他一定会被执行，而且在 jal 的分支延迟时隙中被执行，填充了 jal 的分支延迟时隙，提高了效率。

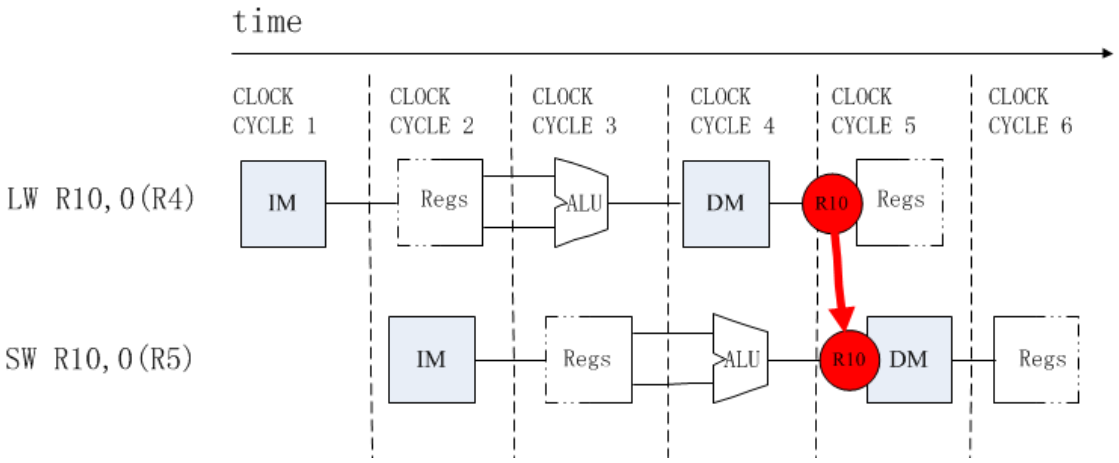
```
00004020 sum:     add $8, $0, $0          # sum function entry
8c890000 loop:    lw $9, 0($4)            # load data
01094020          add $8, $8, $9          # sum
20a5ffff          addi $5, $5, -1         # counter - 1
0005182a          slt $3, $0, $5          # finish?
1460fffb          bne $3, $0, loop        # finish?
20840004          addi $4, $4, 4          # address + 4
03e00008          jr $ra                  # return
01001025          or $2, $8, $0           # move result to $v0
00000000          nop                     # done
```

后半段，我们用相同的方法表示跳转指令和时隙指令，我们同样调转了 bne 和 addi 的，以及 jr 和 or 指令的顺序，同理，这里的 addi 和 or 在老程序中也是必须执行的，在这里同样他们也会在被执行（不论是否选择跳转），在行文上写在分支语句之后不但没有引起程序错误的风险，反而因为填充了分支指令的延迟时隙提高了程序的运行效率。

2.8 载入字时候存储字

这个是个历史遗留问题，我们在如果在 lw（载入字）之后紧跟着一个 sw(存储

字)，那么存储字的输入就会是载入字的输出字，于是我们需要在数据内存的输出返回给输入的增加一条旁路，如图：



这层转发的实现我们可以看 2.6 节最后的电路图，我们发现 Data Ram 存储器的 din 增加了一个多路选择器，这是就是为了实现上文的。

我们可以继续观察返现，这样的旁路转发只有在一种情况下会用到，首先，当下的命令 sw 使用了 rt 作为寄存器(rt_used 被置位)，而且写回的值和上一条指令的写回地址相同（addr_rt == regw_addr_exe），另外，上一条指令是 load 指令（is_load_exe），而这一条指令是 store 指令(is_store)，外加上一条指令确实有写回的操作（wb_wen_exe）。（我们可以理解为 controller 的信号都是在 id 阶段产生的，所以对他而言，上一条指令应该在 EXE 阶段，根据我们的命名规则，我们在查看上一条指令的信息的时候，应当使用其所在阶段的后缀_xxx 形式，以及 yyy_exe）

最后的 fwd_m 信号置 1 的条件为：

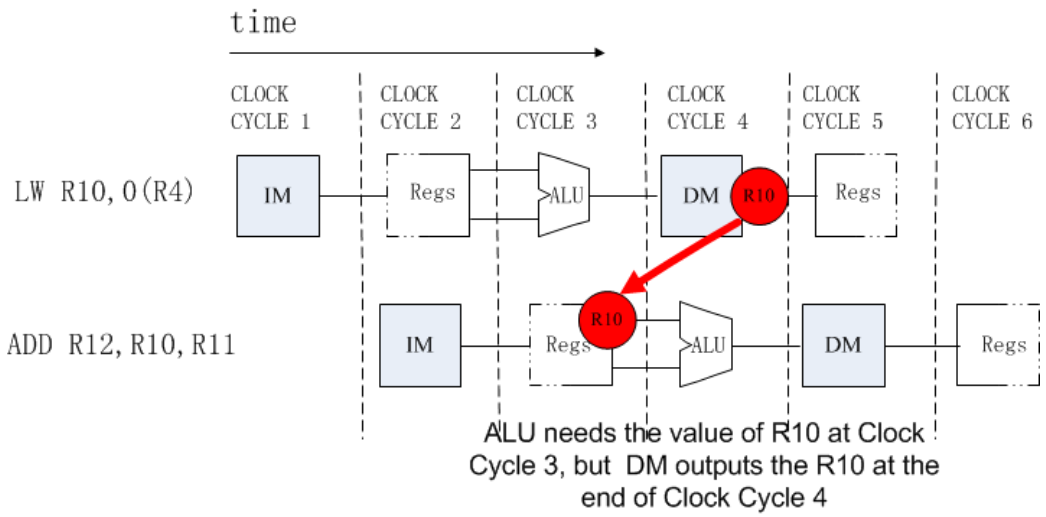
```
if (rt_used && regw_addr_exe == addr_rt && wb_wen_exe && is_load_exe &&
 is_store) begin
```

2.9 旁路无法解决数据冒险的情况

这个也是个历史遗留问题，我们在上节中曾经提到有一种问题，即 ALU 运算在 LW 运算之后，单纯使用旁路，载入字（lw 运算）的结果至少要在 MEM 结果的末尾才能产生，而 ALU 进行运算的时钟周期是会在 EXE 开始时，也就是说同一个时钟周期，数字运算单元必须先等待将其后的内存根据地址输出读取内容，将此内容作为输入，再完成 ALU 运算，最后输出结果，这个过程在上个实验中是可以实

现的，因为一个时钟周期很长，我们可以完全可以在一个时钟周期中完成这些工作，但是在现实中，我们必然不会允许时钟周期过长，因此，我们必须引入停顿，防止程序因为数据冒险而崩溃。

如图所示：



stall 只需要给 if, id 寄存器置 disable 信号，然后重置 EXE 阶段寄存器的值即可，下一个时钟周期，if 和 id 的使能信号都会被置起正常，使得 if, id 都可以正常运作，然后 EXE 会得到 id 提供的，相应寄存器的值。

和 2.9 的情况类似，这里我们也需要保证这条指令的 rs（或者 rt，不失一般性，这里以 rs 为例）被使用了，（rs_used 信号置位），而且和上一条指令的返回值一致（addr_rs == regw_addr_exe），同时上一条指令确实需要写回（wb_wen_exe 置位），另外，如果是 rs 和上一条指令的写回寄存器冲突，那么无论是 R 类型指令还是存储字都需要一步运算，于是都是直接停顿，load_stall 信号置位，但是如果是 rt 寄存器，那么需要分情况，如果是 R 类型指令，依旧需要计算，选择停顿，如果是 sw,那么就和 2.8 完全一致，载入字 lw 的输出成为存储字 sw 的输入，调用 2.8 的转发方案即可，所以，综上，rt 寄存器需要排除本条寄存器是 store 的情况，亦即增加项（~is_store）

最后 load_stall 置 1 的条件为：

```
    if((rs_used && regw_addr_exe == addr_rs && wb_wen_exe && is_load_exe
)
            || (rt_used && regw_addr_exe == addr_rt && wb_wen_exe && is
_load_exe && ~is_store)) begin
                load_stall = 1;
```

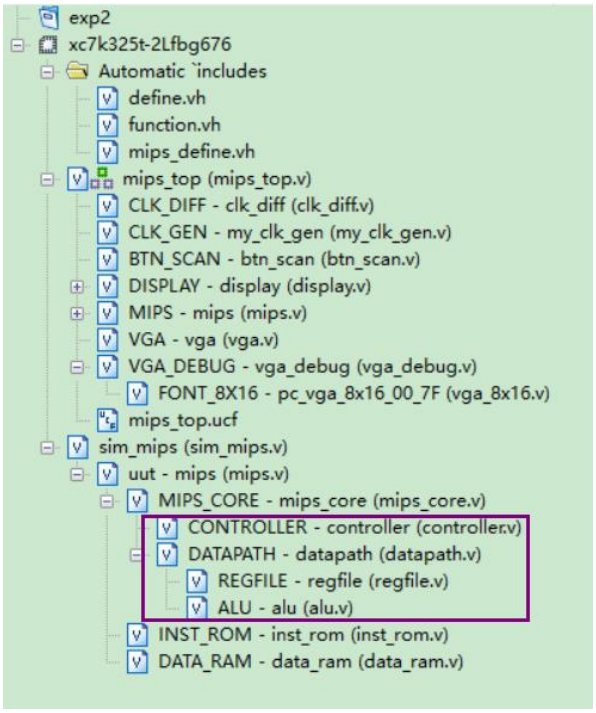
```
end
```

2.10 我们的主要任务

1. 对照电路图修改 datapath，使得旁路单元提前一个周期实现
2. 修改 controller 的值给出控制信号 fwd_m 和 load_stall.

三、 实验过程和数据记录

实验的文件结构如下：



Automatic includes 文件夹下定义了我们可以在代码中使用的头文件；

mips_top.v 是实验的主要工程，实现了对于 MIPS 指令的模拟，clk_diff 和 clk_gen 模块用于 CPU 内置时钟的信号生成的处理，便于各个进程的同步。Btn_san 是处理输入信号的模块，时限完成的 mips_top 模块将烧录到实验室的 sword 开发板上，通过按钮的控制反映 CPU 的计算情况，信息通过 3 个模块输出，分别是 DISPLAY 模块和 VGA，VGA_DEBUG 模块，mips_top.ucf 规定了电路的引脚，使得烧录之后的程序可以嵌入开发板；

sim_mi.v 实际上是一个测试用的工程文件，用于在开发者的电脑上，用 Xilinx ISE 仿真烧录之后的输入输出过程，uut 文件指定了我们的模拟方式（通过内置时钟不断读取 inst_rom 和 data_ram 中的信息，并且把相关的信号反馈到电路图上），仿真检测的对象自然是 mips_core，这个 mips_core 核上文中 mips_top 模块中的 mips_core 是同一个文件。

我们需要完成 mips_core 中的代码补全。mips.v 文件划分为 MIPS_CORE 和 inst_rom，data_ram，完成了二者（CPU 和内存）的交互，mips_core.v 划分为 controller.v 和 datapath.v，完成了二者（控制器和数据通路）的交互，我们重点需要完成 controller.v 和 datapath.v 的代码补全。

1. mips_core 新增信号：

在本次实验中 CPU datapath 首先需要给出 rs_rt_equal 信号（rs 和 rt 是否相等）给 controller，使得 controller 可以快速判断是否需要跳转，而 controller 需要给出信号 fwd_m 指示 datapath 该如何完成 2.9 所言的转发。

源码中对于新增单元使用了注释

```
//exp5 added:
```

字样，便于以后的版本管理。

2. CPU Datapath：

CPU 首先需要做的是将 bypass unit（旁路单元）提前到 id 阶段，如下：

```
//new bypass unit:
    always @(*) begin
        fwda_id = data_rs_exe;
        fwdb_id = data_rt_exe;
        case (exe_fwd_a_ctrl)
            ID_A_FWD_ALUOUT: fwda_id = alu_out_exe;//0
            ID_A_FWD_MEMIN: fwda_id = alu_out_mem;//1
            ID_A_FWD_MEMOUT: fwda_id = mem_din;//2
            ID_A_FWD_RS: fwda_id = data_rs;//3
        endcase
        case (exe_fwd_b_ctrl)
            ID_B_FWD_ALUOUT:fwdb_id = alu_out_exe;//0
            ID_B_FWD_MEMIN: fwdb_id = alu_out_mem;//1
            ID_B_FWD_MEMOUT: fwdb_id = mem_din;//2
            ID_B_FWD_RT: fwdb_id = data_rt;//3
        endcase
    end
    assign rs_rt_equal = (fwda_id == fwdb_id);////judge whether BEQ or
BNE
```

注意我们和上次 bypass unit 代码的区别：

```
    always @(*) begin
```

```
        fwda_exe = data_rs_exe;
        fwdb_exe = data_rt_exe;
        case (exe_fwd_a_ctrl)
            EXE_A_FWD_ALUOUT: fwda_exe = alu_out_mem;//
            EXE_A_FWD_MEMOUT: fwda_exe = mem_din;//1
            EXE_A_FWD_WB: fwda_exe = regw_data_wb;//2
            EXE_A_FWD_RS: fwda_exe = data_rs_exe;//3
        endcase
        case (exe_fwd_b_ctrl)
            EXE_B_FWD_ALUOUT:fwdb_exe = alu_out_mem;//
            EXE_B_FWD_MEMOUT: fwdb_exe = mem_din;//1
            EXE_B_FWD_WB: fwdb_exe = regw_data_wb;//2
            EXE_B_FWD_RT: fwdb_exe = data_rt_exe;//3
        endcase
    end
```

可以看到，相同的宏，例如 EXE_A_FWD_ALUOUT,在上一次实验中引用的是 alu_out_mem，这一次是 alu_out_exe，通过调用不同阶段的寄存器的值这种技巧实现了旁路来源的提前。

之后 CPU datapath 需要完成的是 MEM 阶段根据 fwd_m 转发的情况来选择合适的值转发，也即：

```
        assign
        mem_ren = mem_ren_mem,
        mem_wen = mem_wen_mem,
        mem_addr = alu_out_mem,
        mem_dout = mem_fwd_m_mem?data_rt_mem:regw_data_wb;////
```

注意 CPU controller 给出的 fwd_m 经过各个寄存器的传递在 mem 阶段已经成为 mem_fwd_m_mem，我们根据 mem_fwd_m_mem 的值选择 mem_dout(内存的输入，相当于 datapath 的输出，故称为 dout)的值，data_rt_mem 是指 mem 阶段的 rt 的寄存器值，而 regw_data_wb 就是写回的值，保存了上条指令（此时在 wb 阶段）的写回值。

2. CPU Controller

这次 controller 的实现较为简单，我们只需要按照 2.8，2.9 的方案在最后加上两个输出信号即可

```
always @(*) begin////PPT27x
        exe_fwd_a_ctrl = ID_A_FWD_RS;
        exe_fwd_b_ctrl = ID_B_FWD_RT;
        //exe_fwd_a_ctrl is parallel to fwd_a_ctrl
```

```
        //so is exe_fwd_b_ctrl is parrel to fwd_b_ctrl
        fwd_m = 1'b0;
        load_stall = 1'b0;
       ......

        if (rt_used && regw_addr_exe == addr_rt && wb_wen_exe && is_loa
d_exe && is_store) begin
            fwd_m = 1;
        end

        if((rs_used && regw_addr_exe == addr_rs && wb_wen_exe && is_loa
d_exe)
            || (rt_used && regw_addr_exe == addr_rt && wb_wen_exe && is
_load_exe && ~is_store)) begin
                load_stall = 1;
            end
    end
```

fwd_m 会传递到 datapath 之中，而 load_stall 会在之后的处理 stall 的代码中用到，如下：

```
always @(*) begin
        if_rst = 0;
        if_en = 1;
        id_rst = 0;
        id_en = 1;
        exe_rst = 0;
        ......

        else if (load_stall) begin
            if_en = 0;
            id_en = 0;
            exe_rst = 1;
        end
    end
```

通过 2.9 描述分方法，我们只需要将 if_en，id_en 和 exe_rst 传递给 datapath 即可，而这些已经在 mips_core 中实现了。

四、 实验结果分析

仿真结果：

我们在图片右侧的紫框中找到相应的芯片元件，拖入左侧的图中就可以得到输入输出的信号波形图，注意设定仿真时长，并且显示方式为 16 进制(便于阅读)。

主要观察的波形是 regfile 和 inst 信号的输出，通过观察 inst 信号我们可以确定跳转

的指令是否正确，通过观察 regfile 我们可以确定写回的信号是否正确。

我们测试用的程序如下：

```
00000824 main:      and $1, $0, $0       # address of data[0]
34240050            ori $4, $1, 80       # address of data[0]
0c00000a            jal sum              # call function
20050004 call:      addi $5, $0, 4       # counter
ac820000 return:    sw $2, 0($4)         # store result
8c890000            lw $9, 0($4)         # check sw
ac890004            sw $9, 4($4)         # store result again
01244022            sub $8, $9, $4       # sub: $8 <= $9 - $4
08000008 finish:    j finish             # dead loop
00000000            nop                  # done

00004020 sum:       add $8, $0, $0       # sum function entry
8c890000 loop:      lw $9, 0($4)         # load data
01094020            add $8, $8, $9       # sum
20a5ffff            addi $5, $5, -1      # counter - 1
0005182a            slt $3, $0, $5       # finish?
1460fffa            bne $3, $0, loop     # finish?
20840004            addi $4, $4, 4       # address + 4
03e00008            jr $ra               # return
01001025            or $2, $8, $0        # move result to $v0
00000000            nop                  # done
```

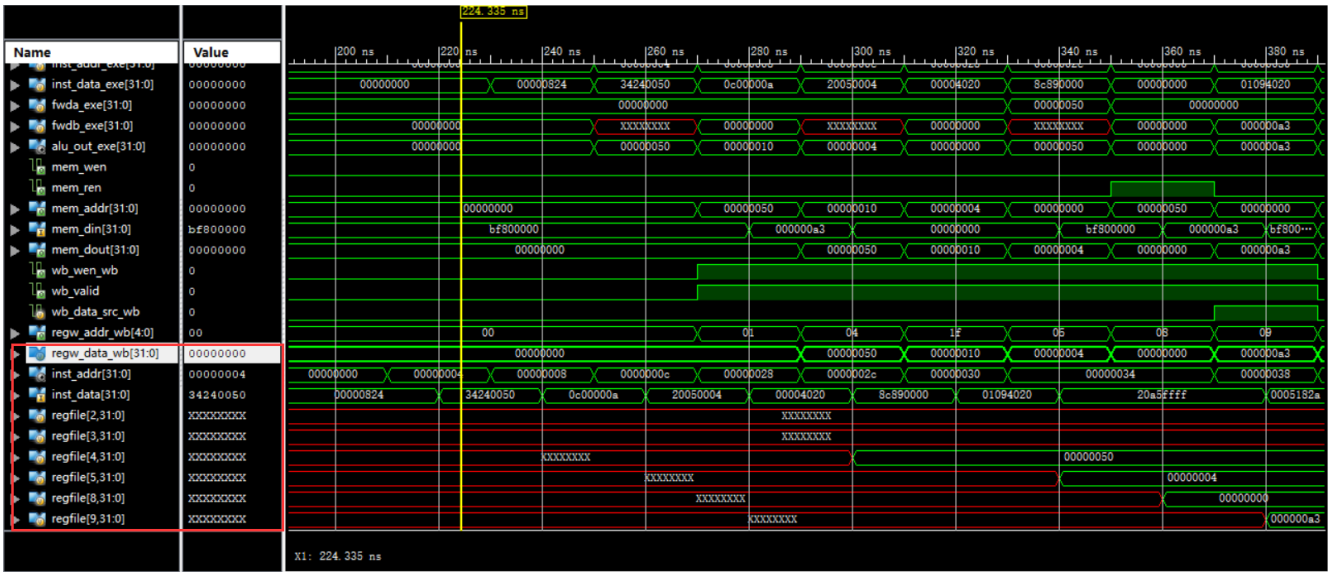
最右侧的 16 进制数是机器码，直接存放在 inst_rom.hex 中，被 CPU 读取。左侧的内容是机器码翻译之后的汇编语言，以及汇编语言的语义（用注释表示）

设定 1 号和 4 号寄存器的数据之后，我们调用函数 sum 将 80 号寄存器的值写入 9 号寄存器，注意我们同样根据 MIPS 规则将每个寄存器存入 32 位（4 字节）的值，于是 80 号相当于第 21 个字（因为 0 号是第一个字），查找 data_ram.hex 文件可以找到他的初始值是 a3，之后 sum 函数给 5 号寄存器不断减 1（5 号初值为 4），同时 4 号寄存器不断加 4（这样每次 9 号寄存器读取出来的值都是内存中下一个字的值）。8 号寄存器则不断累加 9 号寄存器的值，在 5 号小于零前不断循环，最终。将这个值返回给 2 号寄存器。此时四号寄存器已经是 0x60(16 进制的 60 就是十进制的 96， = 80 + 4 * 4)，我们将累加的结果从 2 号寄存器存入内存中的第 25 个字，之后重新读出为 9 号寄存器的值，其值为 0x258，然后我们再减去 4 号寄存器的值 0x60,得到的结果就是八号寄存器的值，应当为 0x1f8.

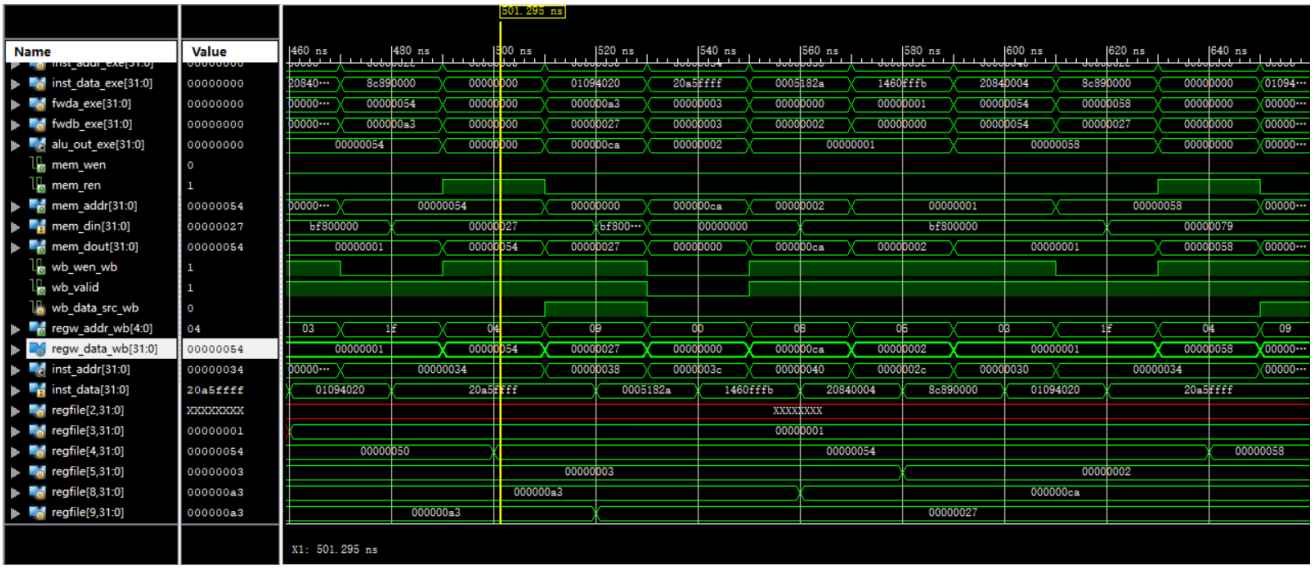
于是，假若 8 号寄存器的最终值为 0x1f8，4 号为 0x60,9 号为 0x258，我们就有很大把握这个 CPU 是正确的。

我们还可以通过 inst 寄存器的输出判断代码的执行顺序，根据汇编语言的逻辑，我们首先从 00000824 顺序执行到 0c00000a（jal 指令），之后跳转到 00004020(sum 位置)执行到 1460ffa(bne),之后跳转为 8c890000(loop 位置),反复执行 4 次，第 4 次到达 1460ffa 之后从 01001025 向下，在 03e00008（jr 指令）出跳转回主程序 ac820000(retrun 位置)最后顺序执行到底，在 0800008（finish 位置）重复跳转到自身，执行死循环。

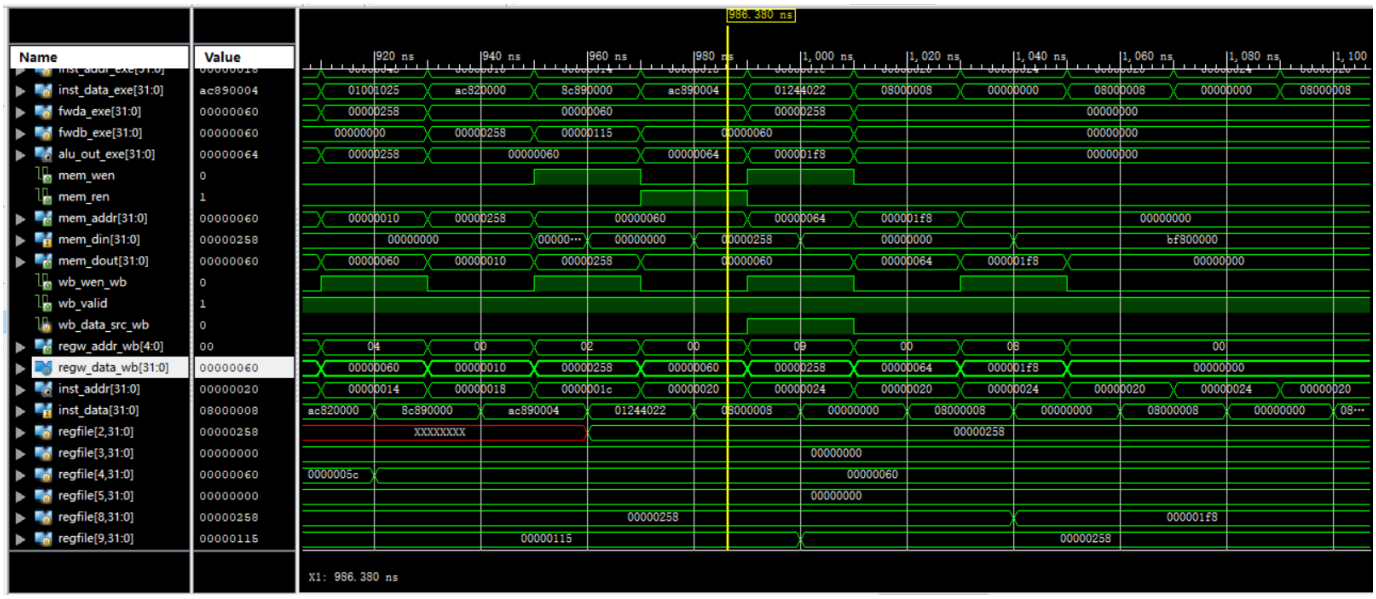
Inst 部分的波形和 Regfile 部分的波形【同一个图】（初始）：



（循环部分）



（结尾部分）



因为在 sword 开发板上的已经通过验收了，所以我们这里的展示从略

在 sword 开发板上的仿真结果（最终结果）：

下文节选一些中间过程


REGS-00 00000000
REGS-01 00000000
REGS-02 00000258
REGS-03 00000000
REGS-04 0000005B
REGS-05 00000004
REGS-06 00000000
REGS-07 00000000
REGS-08 000000A3
REGS-09 000000A3
REGS-0A 00000000
REGS-0B 00000000
REGS-0C 00000000
REGS-0D 00000000
REGS-0E 00000000
REGS-0F 00000000
REGS-10 00000000
REGS-11 00000000
REGS-12 00000000
REGS-13 00000000
REGS-14 00000000
REGS-15 00000000
REGS-16 00000000
REGS-17 00000000
REGS-18 00000000
REGS-19 00000000
REGS-1A 00000000
REGS-1B 00000000
REGS-1C 00000000
REGS-1D 00000000
REGS-1E 00000000
REGS-1F 00000010
IF-ADDR 00000000
IF-INST 0C00000A
ID-ADDR 00000004
ID-INST 34240050
EX-ADDR 00000000
EX-INST 00000024
MM-ADDR 00000000
MM-INST 00000000
RS-ADDR 00000001
RS-DATA 00000000
RT-ADDR 00000004
RT-DATA 00000050
IMMEDAT 0000005B
ALU-AIN 00000000
ALU-BIN 00000000
ALU-OUT 00000000
------- 00000000
FORWARD 00000000
MEMOPER 00001000
MEMADDR 00000000
MEMDATR BF800000
MEMDATW 00000000
WB-ADDR 00000000
WB-DATA 00000000
RESERVE FFFFFFFF
RESERVE FFFFFFFF
RESERVE FFFFFFFF
RESERVE FFFFFFFF
RESERVE FFFFFFFF
RESERVE FFFFFFFF
CP0S-00 00000000
CP0S-01 00000000
CP0S-02 00000258
CP0S-03 00000000
CP0S-04 0000005B
CP0S-05 00000004
CP0S-06 00000000
CP0S-07 00000000
CP0S-08 000000A3
CP0S-09 000000A3
CP0S-0A 00000000
CP0S-0B 00000000
CP0S-0C 00000000
CP0S-0D 00000000
CP0S-0E 00000000
CP0S-0F 00000000
CP0S-10 00000000
CP0S-11 00000000
CP0S-12 00000000
CP0S-13 00000000
CP0S-14 00000000
CP0S-15 00000000
CP0S-16 00000000
CP0S-17 00000000
CP0S-18 00000000
CP0S-19 00000000
CP0S-1A 00000000
CP0S-1B 00000000
CP0S-1C 00000000
CP0S-1D 00000000
CP0S-1E 00000000
CP0S-1F 00000010
RESERVE 00000000
RESERVE 0C00000A
RESERVE 00000004
RESERVE 34240050
RESERVE 00000000
RESERVE 00000024
RESERVE 00000000
RESERVE 00000000
POWER
VOL-
VOL+
MENU
CH-
CH+
MODE


REGS-00 00000000
REGS-01 00000000
REGS-02 00000258
REGS-03 00000001
REGS-04 0000005B
REGS-05 00000002
REGS-06 00000000
REGS-07 00000000
REGS-08 00000143
REGS-09 00000079
REGS-0A 00000000
REGS-0B 00000000
REGS-0C 00000000
REGS-0D 00000000
REGS-0E 00000000
REGS-0F 00000000
REGS-10 00000000
REGS-11 00000000
REGS-12 00000000
REGS-13 00000000
REGS-14 00000000
REGS-15 00000000
REGS-16 00000000
REGS-17 00000000
REGS-18 00000000
REGS-19 00000000
REGS-1A 00000000
REGS-1B 00000000
REGS-1C 00000000
REGS-1D 00000000
REGS-1E 00000000
REGS-1F 00000010
IF-ADDR 00000040
IF-INST 20040004
ID-ADDR 0000003C
ID-INST 1460FFFB
EX-ADDR 00000038
EX-INST 0005102A
MM-ADDR 00000034
MM-INST 20A5FFFF
RS-ADDR 00000003
RS-DATA 00000001
RT-ADDR 00000000
RT-DATA 00000000
IMMEDAT FFFFFFFB
ALU-AIN 00000000
ALU-BIN 00000001
ALU-OUT 00000001
------- 00000000
FORWARD 00000000
MEMOPER 00001000
MEMADDR 00000001
MEMDATR BF800000
MEMDATW 00000143
WB-ADDR 00000000
WB-DATA 00000143
RESERVE FFFFFFFF
RESERVE FFFFFFFF
RESERVE FFFFFFFF
RESERVE FFFFFFFF
RESERVE FFFFFFFF
RESERVE FFFFFFFF
CP0S-00 00000000
CP0S-01 00000000
CP0S-02 00000258
CP0S-03 00000001
CP0S-04 0000005B
CP0S-05 00000002
CP0S-06 00000000
CP0S-07 00000000
CP0S-08 00000143
CP0S-09 00000079
CP0S-0A 00000000
CP0S-0B 00000000
CP0S-0C 00000000
CP0S-0D 00000000
CP0S-0E 00000000
CP0S-0F 00000000
CP0S-10 00000000
CP0S-11 00000000
CP0S-12 00000000
CP0S-13 00000000
CP0S-14 00000000
CP0S-15 00000000
CP0S-16 00000000
CP0S-17 00000000
CP0S-18 00000000
CP0S-19 00000000
CP0S-1A 00000000
CP0S-1B 00000000
CP0S-1C 00000000
CP0S-1D 00000000
CP0S-1E 00000000
CP0S-1F 00000010
RESERVE 00000040
RESERVE 20040004
RESERVE 0000003C
RESERVE 1460FFFB
RESERVE 00000038
RESERVE 0005102A
RESERVE 00000034
RESERVE 20A5FFFF
POWER
VOL-
VOL+
MENU
CH-
CH+
MODE

五、 讨论与心得

之前因为 is_store 没有在每个阶段初始化为 0，所以导致后面有一个寄存器取值是错误的，因为 is_store 一直置 1，导致 fwd_m 一直置一，寄存器的写入的值成为上个阶段的返回值，这个 bug 困扰了我们很久，是通过向其他同学请教才解决的。